Parveen Khurana

124 Followers About Following

# Sigmoid Neuron — Part 2

P Parveen Khurana Jan 3, 2020 · 14 min read

This article covers the content discussed in the Sigmoid Neuron module of the Deep Learning course and all the images are taken from the same module.

In this article, we continue our discussion of the 6 jars of the Machine Learning with respect to the Sigmoid Model.

So far, we have discussed the 4 jars of ML with respect to the Sigmoid Neuron Model. In this article, we will discuss the Learning Algorithm and Evaluation Metric for Sigmoid Neuron.

## Sigmoid: Learning Algorithm

### Intro to Learning Algorithm:

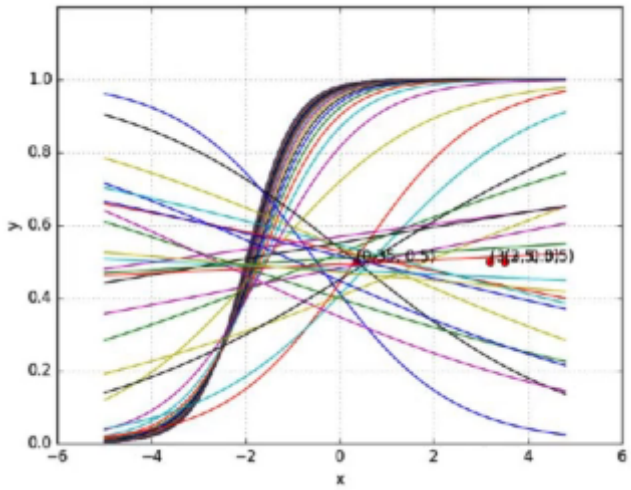
Let's say the below is the input data

| I/P | O/P |
|---|---|
| 2 | 0.047 |
| 3 | 0.268 |
| 4 | 0.73 |
| 5 | 0.952 |
| 8 | 0.999 |

And our model's equation is:

$$h = \frac{1}{1+e^{-(w*x+b)}}$$

The **parameters of the model are w, b** and by changing the values of w, b, we can get different sigmoid functions for example in the below plot, a wide variety of sigmoid functions are plotted each having a different combination of w, b.



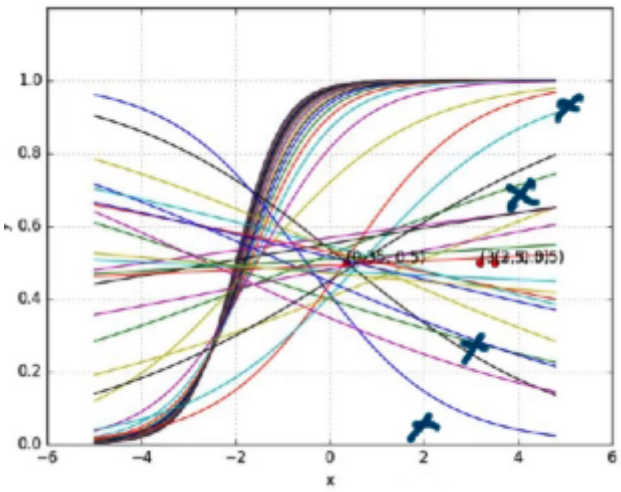

So, by changing w and b we change two things: one is the slope and the second is the position of the sigmoid plot on the x-axis(discussed in this article how the sigmoid plot shifts with the change in the value of the parameters).

**The objective of the learning is to find the parameters of the function in such a way that after plugging in the value of the input, the predicted output is very close to the true output and the loss is minimized.**

If we plot the below input data, it looks like:

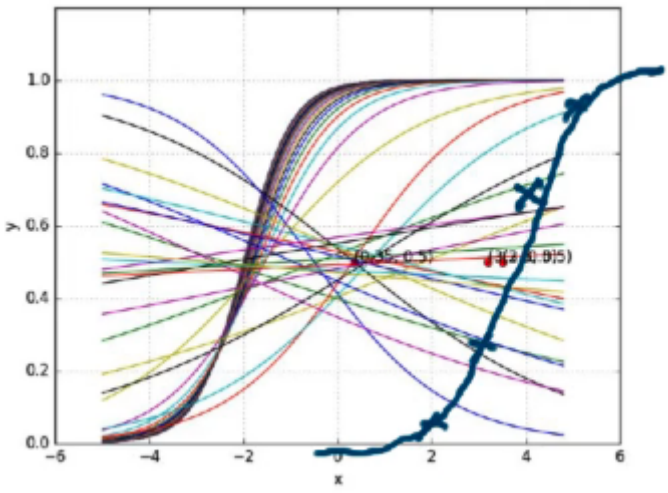
| I/P | O/P |
|---|---|
| 2 | 0.047 |
| 3 | 0.268 |
| 4 | 0.73 |

data




Points in blue cross represent the data in the above table

So, we want the values of w, b in such a way the sigmoid passes through all of the blue cross points in the above image




We start with some random values for w, b; we go over all the training points, compute the loss and try to update the values for w, b in such a way that loss is minimized and we continue this process until we get the desired accuracy.

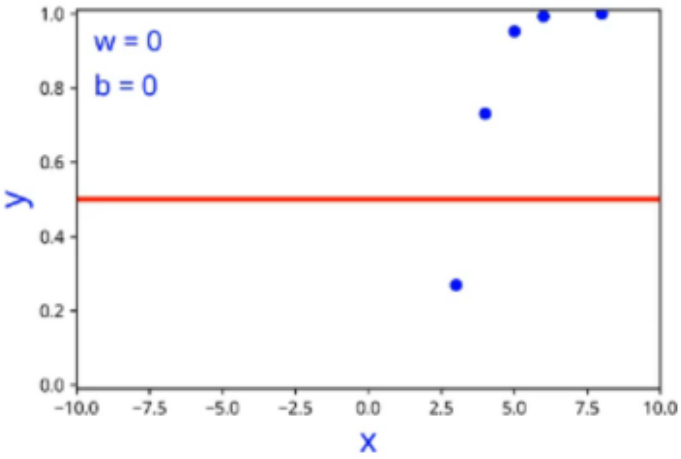
**Learning by Guessing:**

sigmoid with new parameters are closer to the true sigmoid(true output).

We start with w, b as 0.

| I/P | O/P |
|---|---|
| 2 | 0.047 |
| 3 | 0.268 |
| 4 | 0.73 |
| 5 | 0.952 |
| 8 | 0.999 |

$$h = \frac{1}{1+e^{-(w*x+b)}}$$



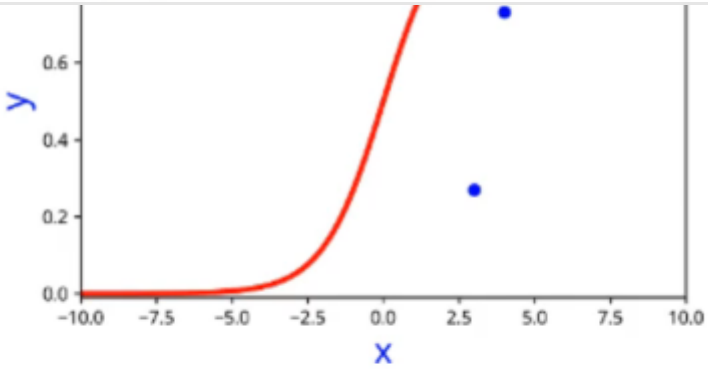

| w | b |
|---|---|
| 0 | 0 |

For w and b as 0, the function would reduce to the value 0.5:

$$\frac{1}{1+e^{-(0)}} = \frac{1}{2}$$

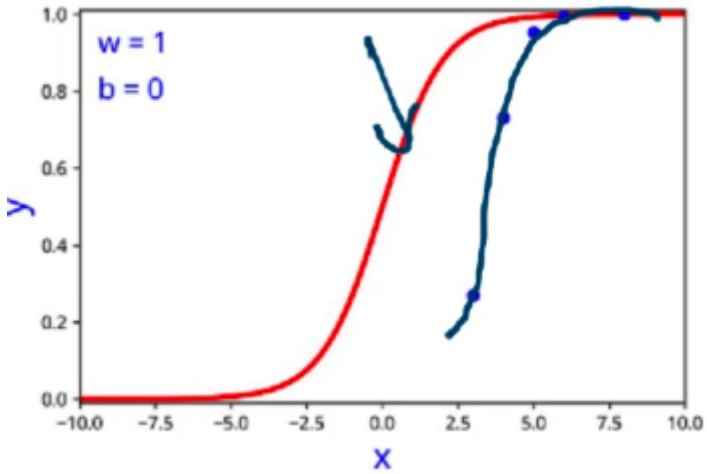
**With w, b as 0, the predicted output is nowhere close to the true output, so we just try to increase w(keeping b as a constant) and see the output:**

$$h = \frac{1}{1+e^{-(w*x+b)}}$$

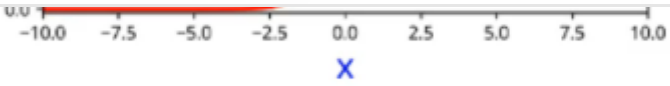



| w | b |
|---|---|
| 1 | 0 |

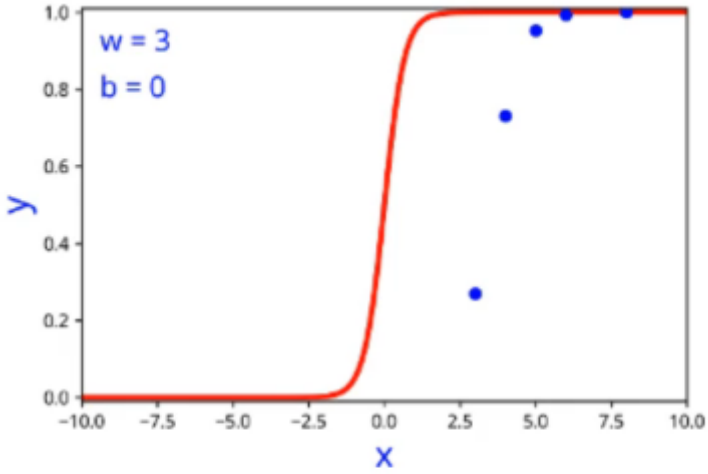
Now we can notice that at least the slope of the above sigmoid function is somewhat better because in the ideal scenario when the plot passes through all the data points, we need a positive slope:




So, we know that increasing w is a step in the right direction, so we increase w a little bit more and see the plot:



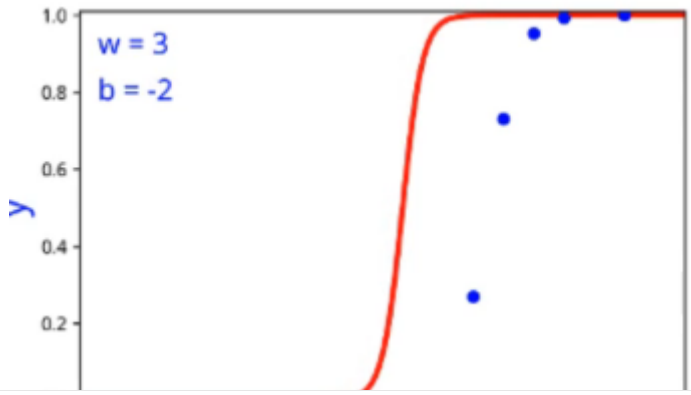

| w | b |
|---|---|
| 2 | 0 |



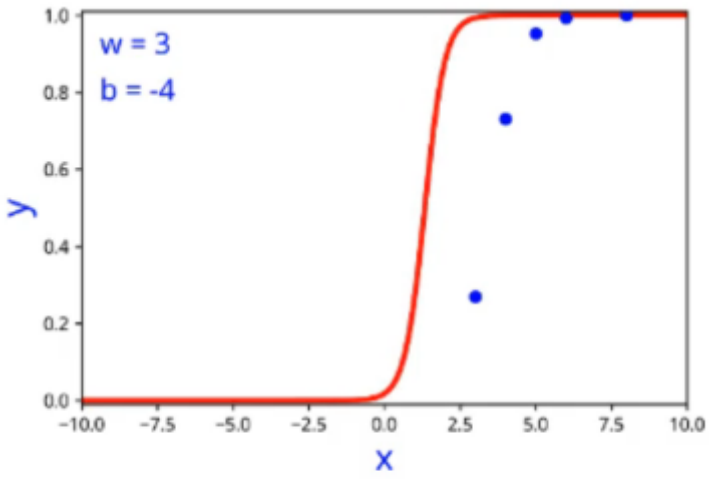

| w | b |
|---|---|
| 3 | 0 |

Let's keep the value of '**w**' as 3 for now and experiment with **b**.

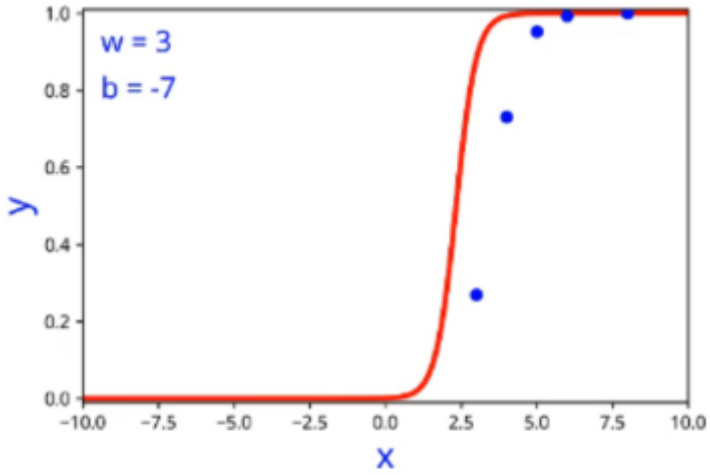
We want the sigmoid plot(with '**w**' as 3) to move towards the right and we know that to shift it towards the right, we need to decrease the value of **b**. So, we have the following:



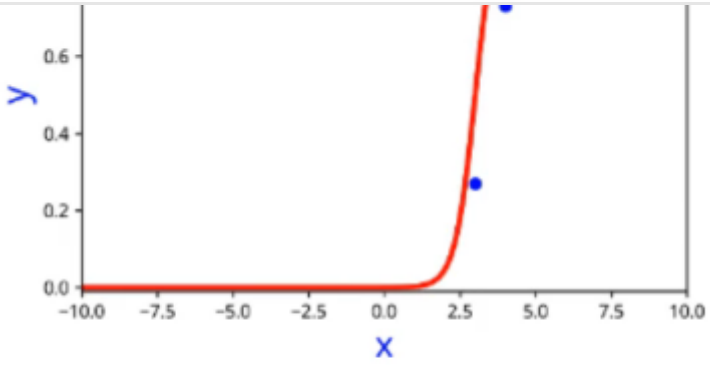
| w | b |
|---|---|
| 3 | -2 |




| w | b |
|---|---|
| 3 | -4 |

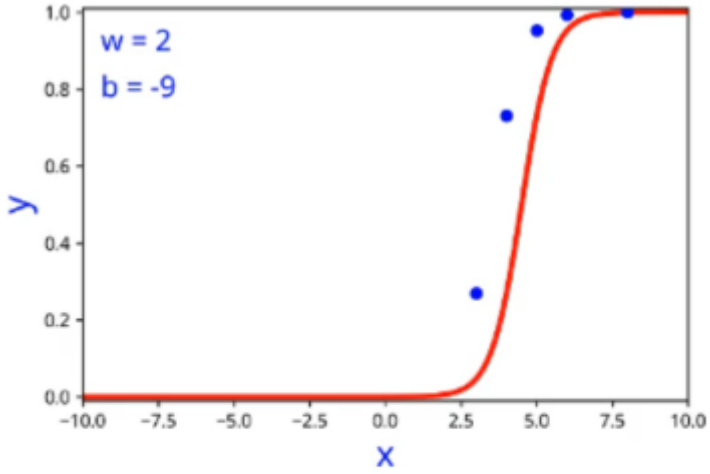



| w | b |
|---|---|
| 3 | -7 |



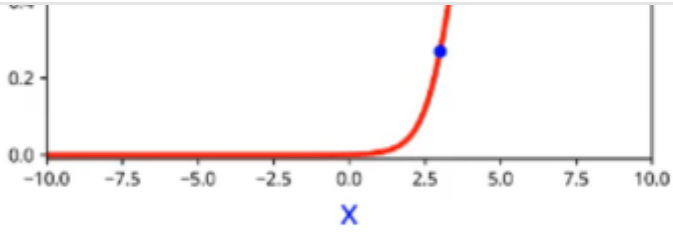

| w | b |
|---|---|
| 3 | -9 |

We are very close to the solution at this point, now we again see if changing **w** helps us here:




| w | b |
|---|---|
| 2 | -9 |

But decreasing w from 3 to 2 and keeping the b same overshot the point, so let's see if it helps to keep w same and decreasing b:




| w | b |
|---|---|
| 2 | -7 |

And now we are able to exactly fit the points.

**We were able to plot this 1 dimensional and see for ourselves how changing the values of the parameters affects the plot but in practice, data would be high dimensional and we would not be able to plot it out and get insights from it that easily.**

General Recipe:

**Initialise** $w, b$

**Iterate over data:**

$$w = w + \Delta w$$
$$b = b + \Delta b$$

**till satisfied**

**Iterate over data:**

$$w = 0$$
$$b = 0$$

In the next iteration, we changed w to 1 and kept b as 0:

**Iterate over data:**

$$1 = 0 + \Delta 1$$
$$0 = 0 + \Delta 0$$

And we kept doing this till we got the exact plot.

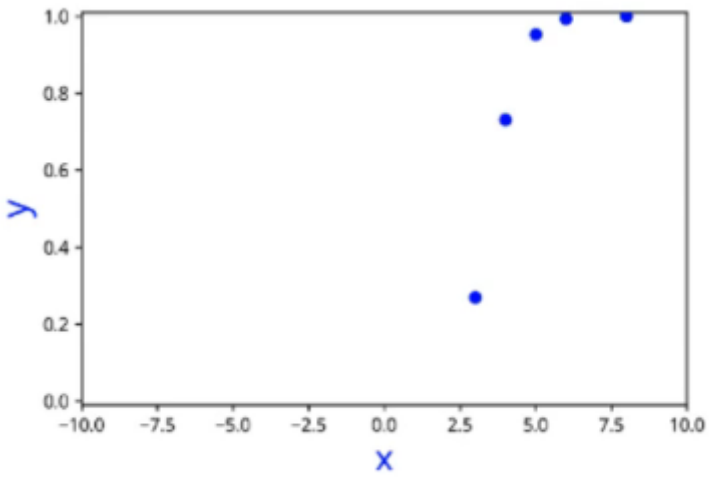
The only problem with this algorithm is that the update values(**$\delta w$ and $\delta b$**)(delta w and delta b) are coming from guesses which we can't do in real-world problems as it might take a very very long time to converge to a solution:

$$\Delta w = some\_guess$$
$$\Delta b = some\_guess$$

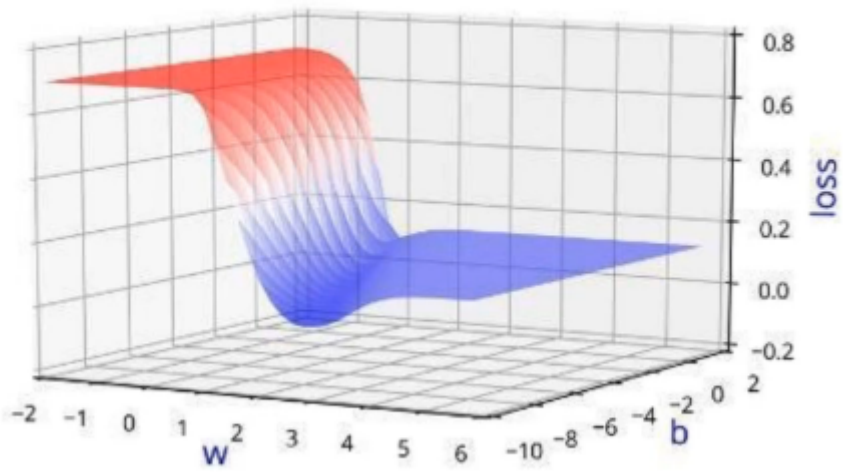
So, what we need is a more principle approach guided by Loss function so that we update the parameters in the right direction and with every update, the loss values decreases.

different parameter combinations and a change in the parameter value changes the predicted output which in turn changes the loss value.

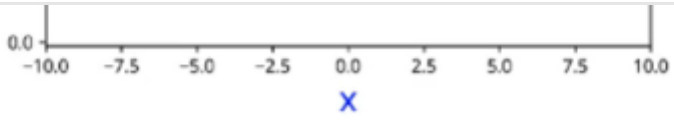
We can also plot out the loss value for different values of w and b:

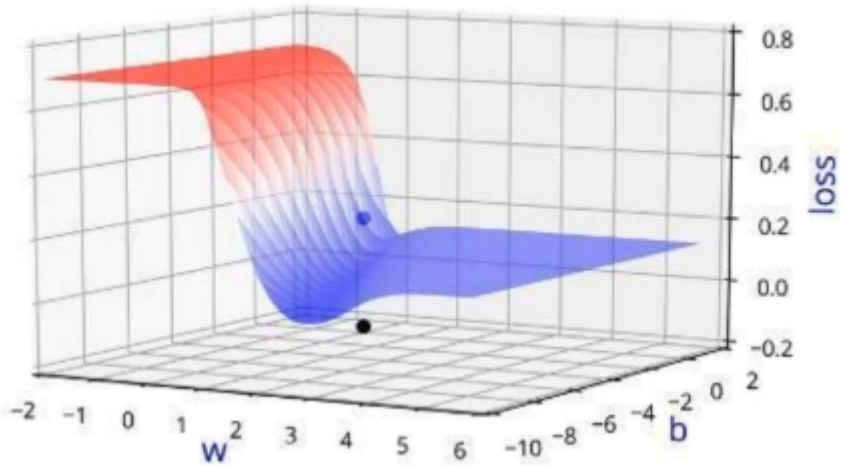



Data

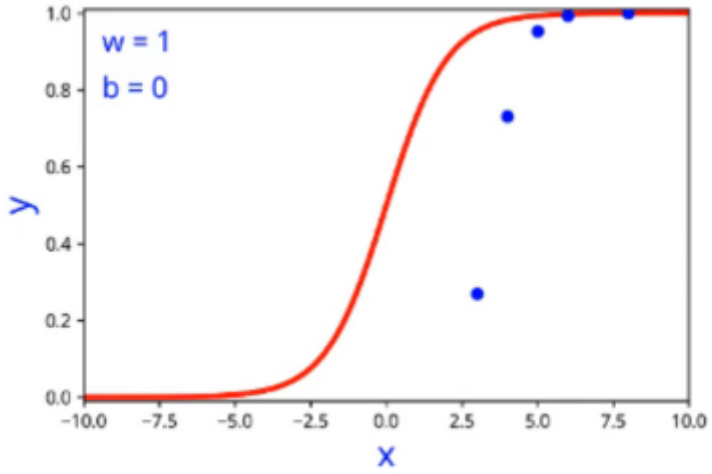


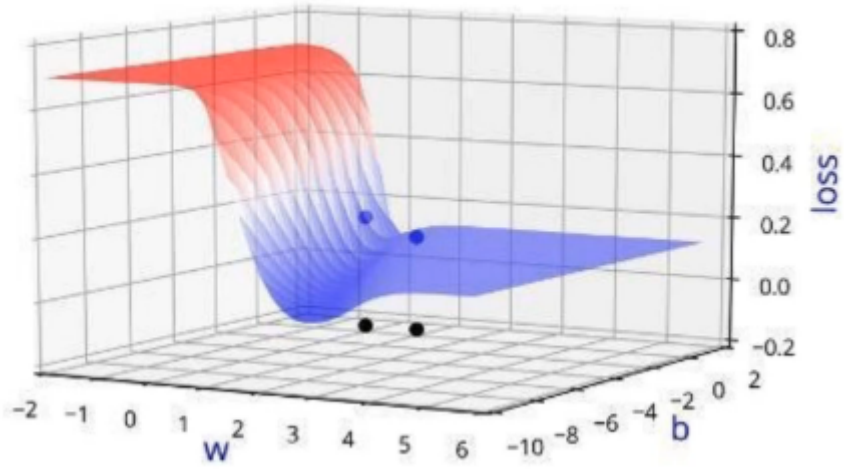

Let's see what we were doing in the guesswork algorithm:

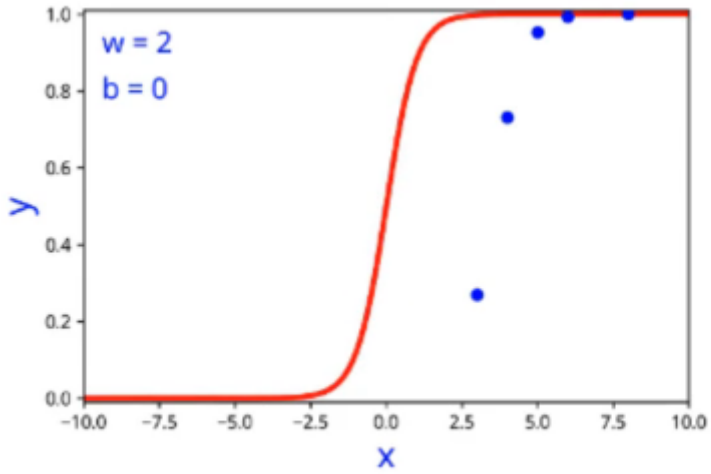


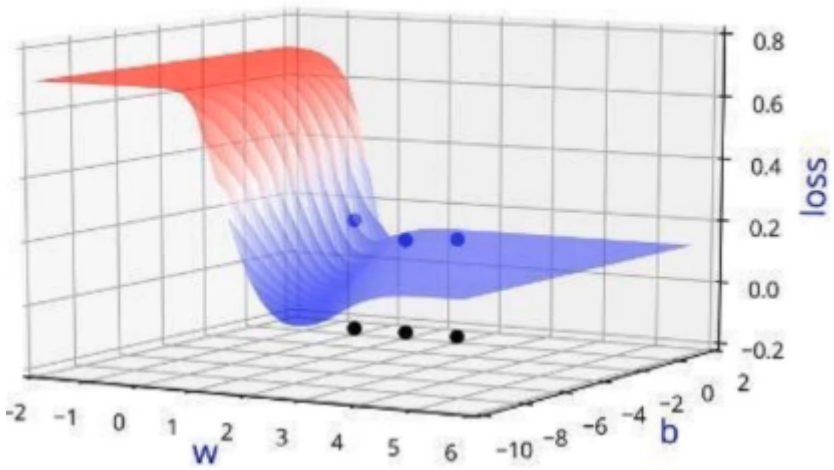




When w = 1 and b = 0, the loss is changed






And in the guesswork algorithm we were randomly changing the values of **w and b** and were randomly moving on the error surface:

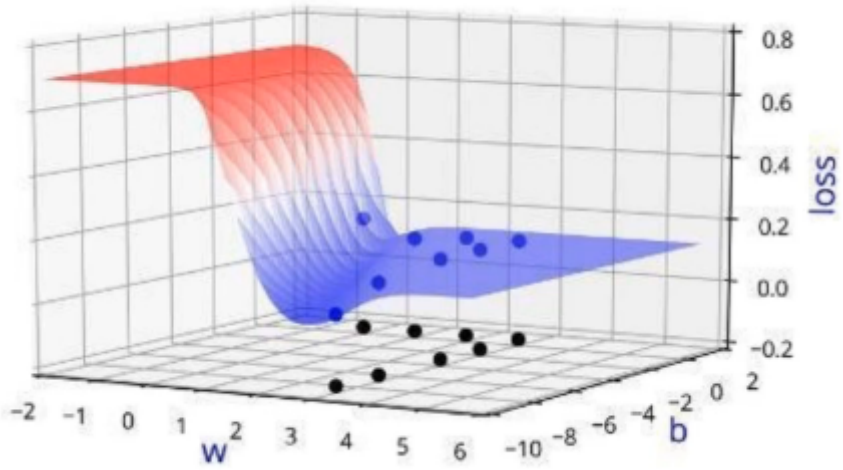
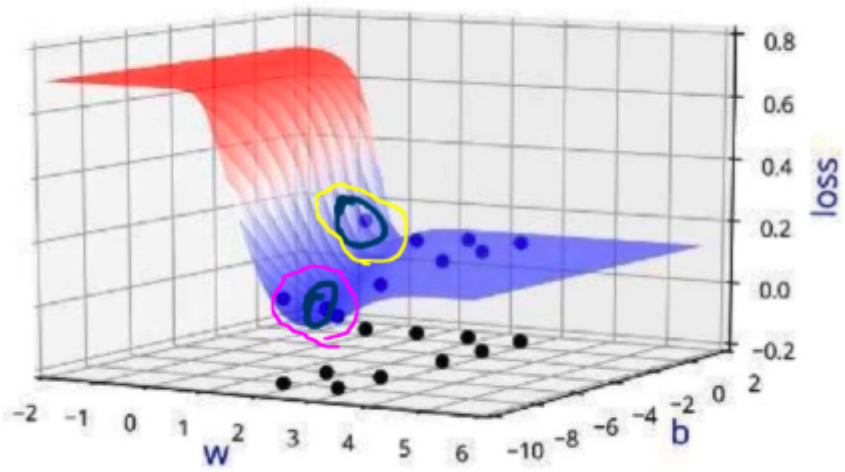


image) such that we don't make any random movements where we are increasing, decreasing w, b and the loss is also increasing, decreasing on the error surface, **we must make movements in such a way that on this error surface we are constantly decreasing the value of the loss function which was not the case with the random guess algorithm.**




So, our main aim as of now is that we need a principled way of changing **w and b** based on the loss function.

"Instead of guessing $\Delta w$ and $\Delta b$, we need a principled way of changing w and b based on the loss function"

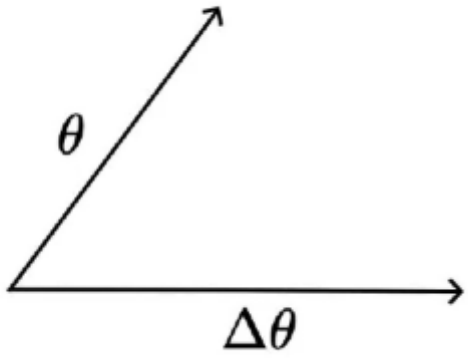
We can consider the parameters as a vector **θ** and then our goal is to find the optimal value of **θ** for which the loss is minimized. And the way we do that is we start with some random value for **θ** and we are going to change it iteratively

$$\theta = [w, b]$$

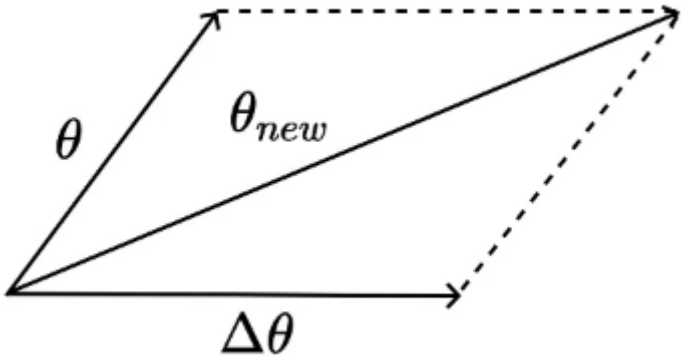
$$\Delta\theta = [\Delta w, \Delta b]$$

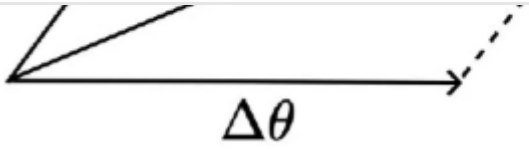
and in every iteration, we are going to make it as the following and we keep updating it till satisfied with the loss value:

Geometrically, we have the **θ** and the change in **θ** as a vector which we can represent as:

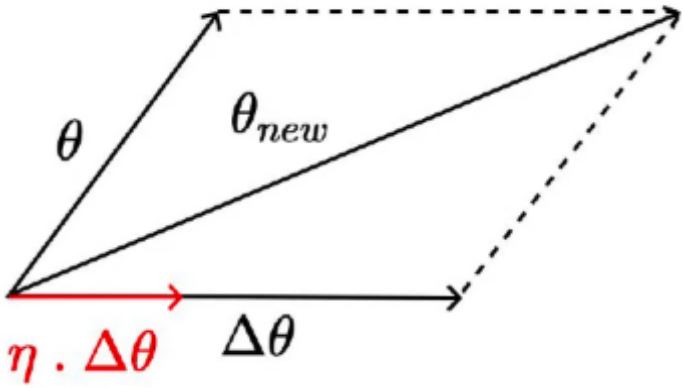



The new value for **θ** can be represented as:

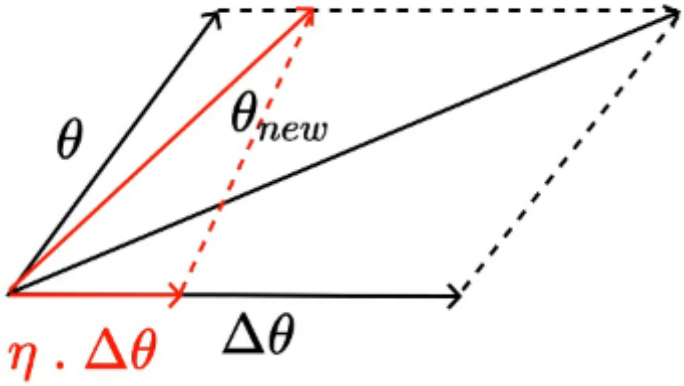



One thing that we note is that there is this large change in **θ**, so instead of updating by delta theta, we can update the **θ** with a smaller quantity(which is going to be in the same direction as delta theta)




And in this case, the new value for **θ** would look like:




would not like to change the parameters drastically.

### What to do to update the parameters?

So, we have the data given to us, we have the model that we have chosen

| I/P | O/P |
|---|---|
| 3 | 0.268 |
| 4 | 0.73 |
| 5 | 0.952 |
| 6 | 0.994 |
| 8 | 0.999 |

$$\hat{y} = \frac{1}{1 + e^{-(wx+b)}}$$

The **way learning proceeds** is that we compute **y_hat(output) for each of the data points and once we have the predicted output for each of the points, we can compute the loss value as**:

$$Loss\mathscr{L}(w,b) = \sum_{i=1}^{5}(y_i - \hat{y}_i)^2$$

And once we have the overall loss value, we can also compute the average loss value. **Now based on the loss value, we want to know the right step(direction of change, the magnitude of change) to change the parameters** and the mathematical details of the same are discussed in this article. There are functions in frameworks like PyTorch, TensorFlow which automatically computes the delta values for the parameters. And once we update the parameters, we again go through the entire data,

accuracy or loss is less than a pre-defined threshold or we can just iterate a pre-fixed number of times or we can compare the parameter's value in two consecutive iterations and if there is not large change between the parameter values, we can stop iterating.

**Initialise** $w, b$

**Iterate over data:**

$compute\ \hat{y}$

$compute\ \mathscr{L}(w, b)$

$$w_{t+1} = w_t - \eta \Delta w_t$$

$$b_{t+1} = b_t - \eta \Delta b_t$$

**till satisfied**

As discussed in the article conveying mathematical details, we can write the partial derivatives with respect to **w and b** for the 5 data points that we have in this case as:

For 5 points,

$$\Delta w = \sum_{i=1}^{i=5} (f(x_i) - y_i) * f(x_i)) * (1 - f(x_i)) * x_i$$

$$\Delta b = \sum_{i=1}^{i=5} (f(x_i) - y_i) * f(x_i)) * (1 - f(x_i))$$

### Writing the Code:

We have two inputs(data points) represented in the form of an array

```
X = [0.5, 2.5]
Y = [0.2, 0.9]
```

We would approximate the relationship between the input and the output using a Sigmoid function.

```
def f(w, b, x):
   #sigmoid with parameters w, b
   return 1.0 / (1.0 + np.exp(-(w*x + b))
```

We have another function to compute the error. This function takes the parameter values as the input, it goes over the data points, compute the predicted output for each of the data points, computes the squared error loss for each of the data points and add to the main loss and then, in the end, it returns this main loss.

```
def error(w, b):
     err = 0.0
     for x, y in zip(X, Y):
        fx = f(w, b, x)
        err += 0.5* (fx - y) ** 2
     return err
```

We can compute the derivative with respect to w and b as:

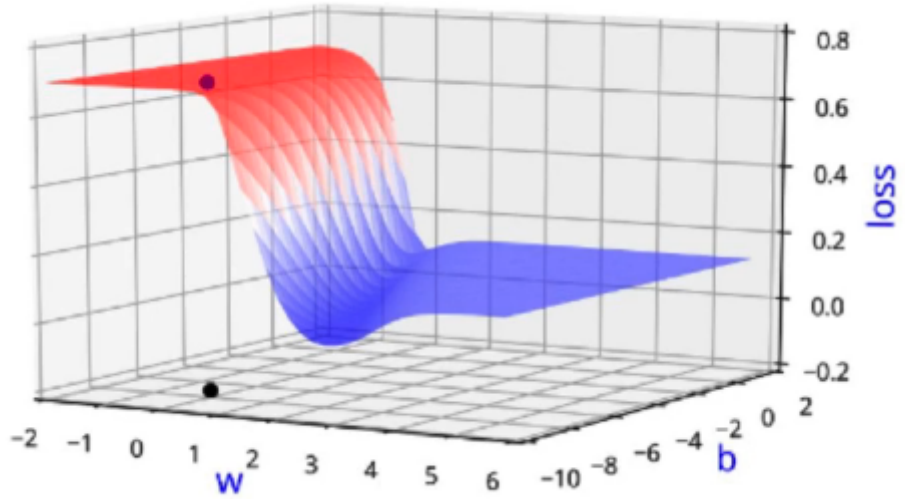
```
def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x
```

Now we look at the main function in which we iteratively go through the data and update the parameters:

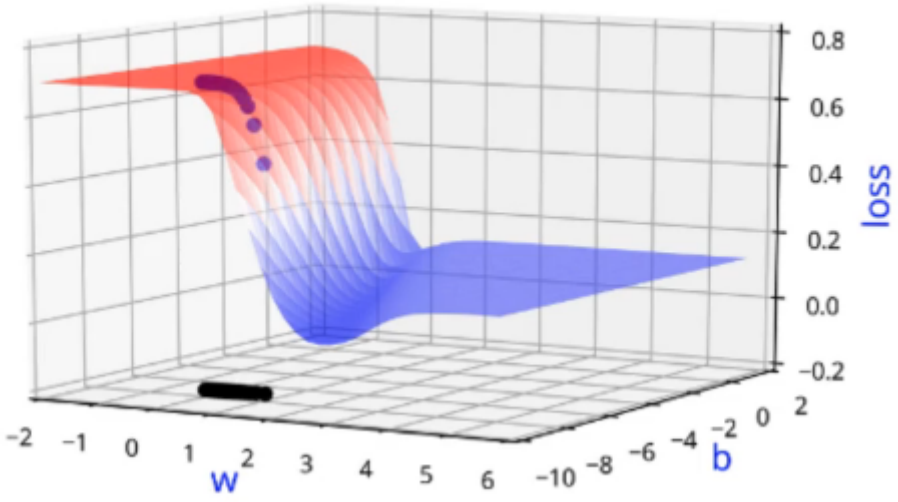
We start with random values for w, b, and we choose some value for learning rate, we set some value for the number of epochs(number of times to iterate over the data) and for each pass, we compute the derivatives and update the weights

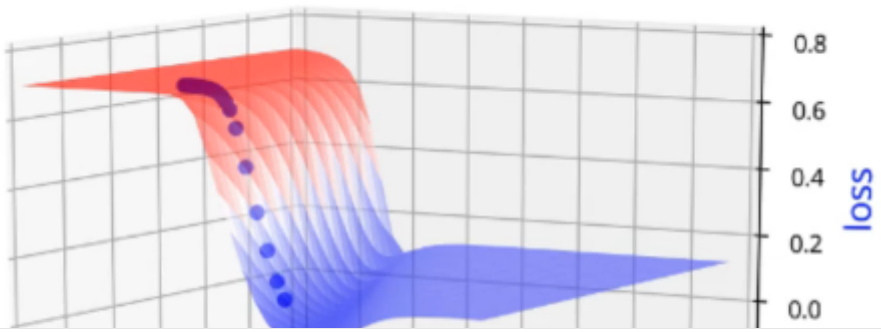
```
def do_gradient_descent():
    w, b, eta = -2, -2, 1.0
    max_epochs = 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

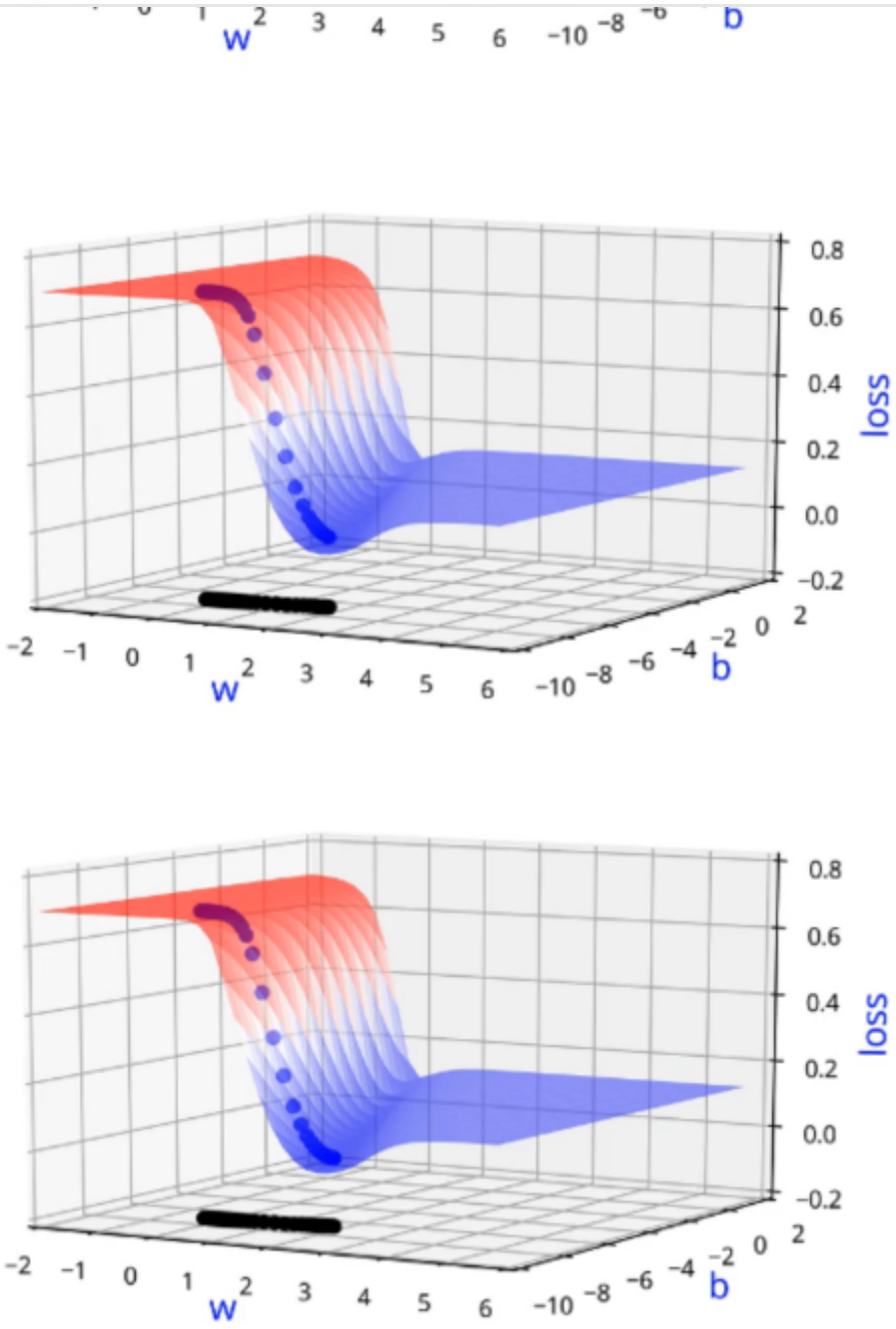
To start with, we have '**w**' as 0 and b as -8




After a few iterations, the situation is like:






The error is constantly decreasing(not jumping up and down like the way it was in random guess algorithm), it is never increasing at any point and after a few iterations, we have reached the dark blue region where the error is close to 0.

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x):
   #sigmoid with parameters w, b
   return 1.0 / (1.0 + np.exp(-(w*x + b))

def error(w, b):
     err = 0.0
     for x, y in zip(X, Y):
        fx = f(w, b, x)
```

```
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta = -2, -2, 1.0
    max_epochs = 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

## Dealing with more than 2 parameters

Let's take the scenario when we have more than 2 parameters or to say more than one input:

| Emergency Room Visits | Narcotics | Pain | Total Visits | Medical Claims | PoorCare |
|---|---|---|---|---|---|
| 0 | 2 | 6 | 11 | 53 | 1 |
| 1 | 1 | 4 | 25 | 40 | 0 |
| 0 | 0 | 5 | 10 | 28 | 0 |
| 1 | 3 | 5 | 7 | 20 | 1 |

Let's say we are trying to predict the output PoorCare using the 5 input features that we have in the above image:

$$z = w_1 * ER\_visits + w_2 * Narcotics + w_3 * Pain + w_4 * TotalVisits + w_5 * MedicalClaims + b$$

The data matrix is usually denoted by X and the output vector by Y.

| | | | | | |
|---|---|---|---|---|---|
| 1 | 1 | 4 | 25 | 40 | 0 |
| 0 | 0 | 5 | 10 | 28 | 0 |
| 1 | 3 | 5 | 7 | 20 | 1 |

Each row in the input and output data matrix/vector refers to one data point.

The number in yellow in the below image:

| Emergency Room Visits | Narcotics | Pain | Total Visits | Medical Claims | PoorCare |
|---|---|---|---|---|---|
| 0 | 2 | 6 | 11 | 53 | 1 |
| 1 | 1 | 4 | 25 | 40 | 0 |
| 0 | 0 | 5 | 10 | 28 | 0 |
| 1 | 3 | 5 | 7 | 20 | 1 |

**we can write it as x12 meaning the 2nd input feature for the first data point/row.**

In general, **we can write it as '$x_{ij}$' where 'i' refers to the row index and 'j' refers to the column index of the matrix.**

So, **for the i'th data item, the way we are going to compute the sum as:**

$$z = w_1 * x_{i1} + w_2 * x_{i2} + w_3 * x_{i3} + w_4 * x_{i4} + w_5 * x_{i5} + b$$

And then we can pass this as the input to the sigmoid function

$$w_5 * MedicalClaims + b$$

$$z = w_1 * x_{i1} + w_2 * x_{i2} + w_3 * x_{i3} + w_4 * x_{i4} + w_5 * x_{i5} + b$$

$$\hat{y} = \frac{1}{1+e^{-z}}$$

$$\hat{y} = \frac{1}{1+e^{-(w_1*x_{i1}+w_2*x_{i2}+w_3*x_{i3}+w_4*x_{i4}+w_5*x_{i5}+b)}}$$

So, we have a total of 6 parameters in this case -> 5 weight terms and 1 bias term. And the way we are going to update all of them is by using the same gradient descent algorithm.

**Initialise** $w_1, w_2, \ldots, w_5, b$

**Iterate over data:**

$$w_1 = w_1 - \eta \Delta w_1$$

$$w_2 = w_2 - \eta \Delta w_2$$

$$\vdots$$

$$w_5 = w_5 - \eta \Delta w_5$$

$$b = b - \eta \Delta b$$

**till satisfied**

If we have one input and one weight parameter, then the derivative of the loss function with respect to that weight is given as:

$$\Delta w = \sum_{i=1}^{m} (f(x) - y) * f(x) * (1 - f(x)) * x$$

respect to first weight/parameter is given as:

$$\Delta w_1 = \sum_{i=1}^{m} (\hat{y} - y) * \hat{y} * (1 - \hat{y}) * x_{i1}$$

So, this value depends on the first column for the **i'th** input and we are going to sum it over all the inputs.

Similarly, we can write:

$$\Delta w = \sum_{i=1}^{m} (f(x) - y) * f(x) * (1 - f(x)) * x$$

$$\Delta w_1 = \sum_{i=1}^{m} (\hat{y} - y) * \hat{y} * (1 - \hat{y}) * x_{i1}$$

$$\Delta w_2 = \sum_{i=1}^{m} (\hat{y} - y) * \hat{y} * (1 - \hat{y}) * x_{i2}$$

$$\Delta w_j = \sum_{i=1}^{m} (\hat{y} - y) * \hat{y} * (1 - \hat{y}) * x_{ij}$$

The same changes we have in the code as:

Earlier we had the function as:

```
def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x
```

And now we have:

```
def grad_w_i(w, b, x, y, i):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x[i]
```

Earlier we had:

```
def f(w, b, x):
    #sigmoid with parameters w, b
    return 1.0 / (1.0 + np.exp(-(w*x + b))
```

Now we have:

```
def f(w, b, x):
    #sigmoid with parameters w, b
    return 1.0 / (1.0 + np.exp(-(np.dot(w, x) + b))
```

There will changes in the gradient descent function also where instead of updating one weight, we now have to update all the weights.

## Evaluation:

So, we have the data given to us, the task is to predict some value between 0 to 1 because we are trying to regress a probability, the model that we choose is the Sigmoid model and we have the Loss function as the squared error loss function. Now we will see how to evaluate the model for the test data.

Training data

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Launch (within 6 months)** | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| **Weight** | 0.19 | 0.63 | 0.33 | 0.99 | 0.36 | 0.66 | 0.1 | 0.70 | 0.48 |
| **Screen size** | 0.64 | 0.87 | 0.67 | 0.88 | 0.7 | 0.91 | 0.04 | 0.98 | 0.47 |
| **dual sim** | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| **Internal memory (>= 64 GB, 4GB RAM)** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **NFC** | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| **Radio** | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| **Battery** | 0.36 | 0.51 | 0.36 | 0.97 | 0.34 | 0.67 | 0 | 0.57 | 0.43 |
| **Price** | 0.09 | 0.63 | 0.41 | 0.19 | 0.06 | 0 | 0.72 | 0.94 | 1 |
| **Like (y)** | **0.9** | **0.3** | **0.85** | **0.2** | **0.5** | **0.98** | **0.1** | **0.88** | **0.23** |

$$y = \frac{1}{}$$

$$Loss = \sum_{i=1}^{n}(y - \hat{y})^2$$

We have the test data which we can pass through our model and get the predicted output and compare the true output with the predicted output

Test data

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 0.2 | 0.73 | 0.6 | 0.8 |
| 0.2 | 0.7 | 0.8 | 0.9 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |
| 0.83 | 0.96 | 0.9 | 0.2 |
| 0.34 | 0.4 | 0.6 | 0.1 |
| **0.17** | **0.56** | **0.3** | **0.4** |
| **0.24** | **0.67** | **0.9** | **0.3** |


We can compute the Root Mean Squared Error(RMSE) which is typically used for regression problems.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y - \hat{y})^2}$$

The smaller the RMSE the better.

And if we are doing the classification task, and we have to give a discreet output either 0 or 1, we can choose a threshold value and map the predicted output then to either 0 or 1.

Training data

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Launch (within 6 months) | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **dual sim** | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| **Internal memory (>= 64 GB, 4GB RAM)** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **NFC** | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| **Radio** | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| **Battery** | 0.36 | 0.51 | 0.36 | 0.97 | 0.34 | 0.67 | 0 | 0.57 | 0.43 |
| **Price** | 0.09 | 0.63 | 0.41 | 0.19 | 0.06 | 0 | 0.72 | 0.94 | 1 |
| **Like (*y*)** | **1** | **0** | **1** | **0** | **1** | **1** | **0** | **1** | **0** |

$$y = \frac{1}{1+e^{-(w^T x+b)}}$$

$$Loss = \sum_{i=1}^{n} (y - \hat{y})^2$$

Test data

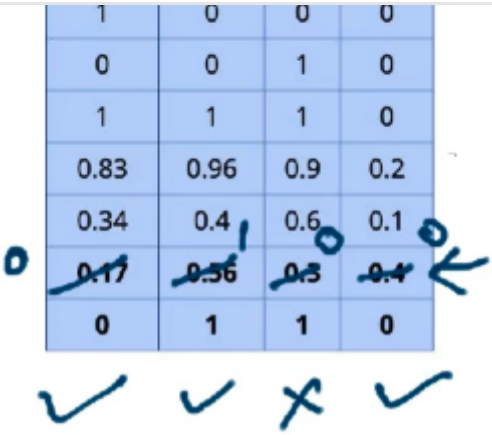
| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 0.2 | 0.73 | 0.6 | 0.8 |
| 0.2 | 0.7 | 0.8 | 0.9 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |
| 0.83 | 0.96 | 0.9 | 0.2 |
| 0.34 | 0.4 | 0.6 | 0.1 |
| **0.17** | **0.56** | **0.3** | **0.4** |
| **0** | **1** | **1** | **0** |

Threshold=0.5

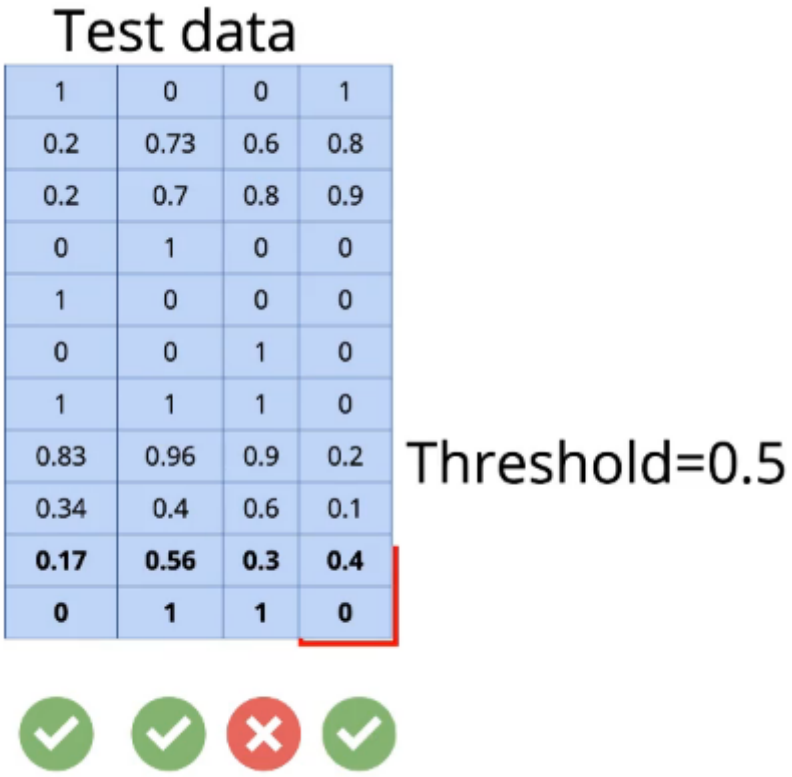
We will take any value less than 0.5 as 0 and value greater than 0.5 as 1.

Test data

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 0.2 | 0.73 | 0.6 | 0.8 |



| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |
| 0.83 | 0.96 | 0.9 | 0.2 |
| 0.34 | 0.4 | 0.6 | 0.1 |
| 0.17 | 0.56 | 0.3 | 0.4 |
| 0 | 1 | 1 | 0 |



Test data

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 0.2 | 0.73 | 0.6 | 0.8 |
| 0.2 | 0.7 | 0.8 | 0.9 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |
| 0.83 | 0.96 | 0.9 | 0.2 |
| 0.34 | 0.4 | 0.6 | 0.1 |
| **0.17** | **0.56** | **0.3** | **0.4** |
| **0** | **1** | **1** | **0** |

Threshold=0.5

And we can compute the accuracy using the standard formula that we have:

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

## Summary:

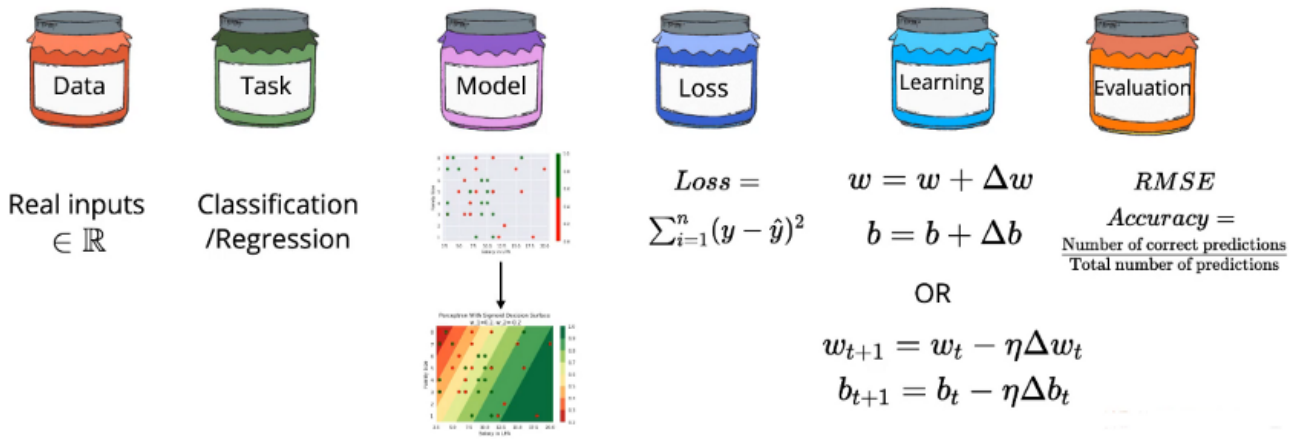
**Data:** We are dealing with Real numbered inputs.

**Model:** Our approximation between the input and the output is a non-linear function which helps us to get the graded output which we can express as a probability.

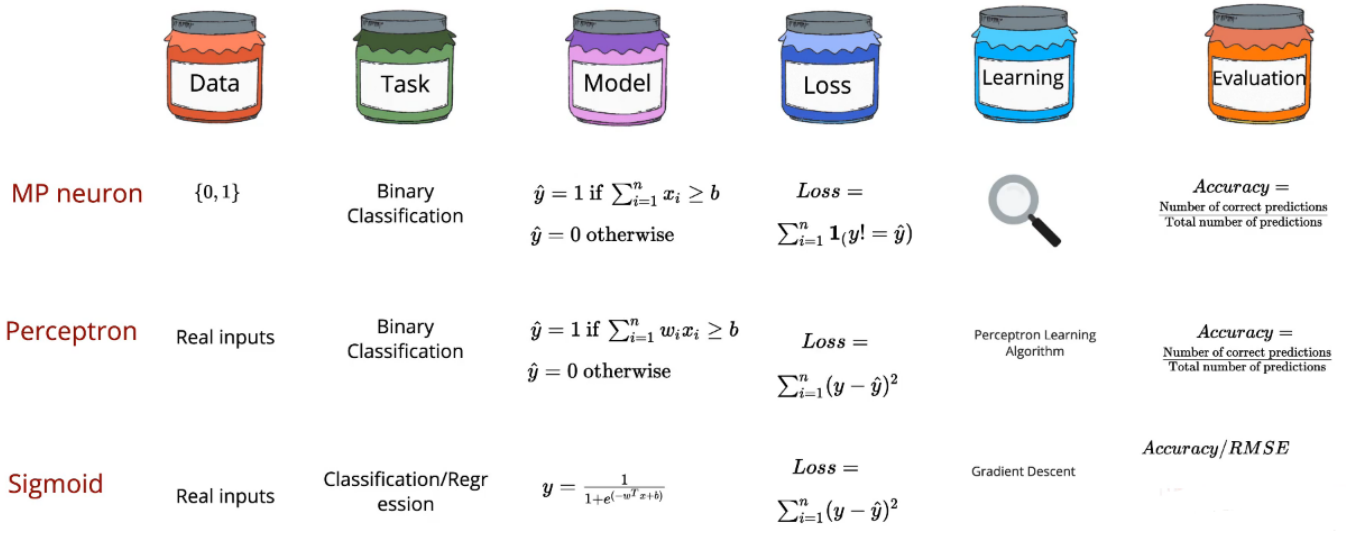
**Loss**: We are still using the Squared Error loss function.

**Learning Algorithm:** We used the Gradient Descent algorithm which ensured that every step that we take with the updating of parameters, it points in the right direction.

**Evaluation**: We can use the RMSE metric for the regression problem and we can use the Accuracy metric for classification tasks.




## Comparison of MP Neuron, Perceptron and Sigmoid in terms of 6 Jars



Open in app

About Write Help Legal

Get the Medium app